InSAR Scientific Computing Environment

Eric Gurrola[1], Paul A. Rosen[1], Gian Franco Sacco[1], Walter Seliga[1], Howard Zebker[2], Mark Simons[3], David Sandwell[4]

[1]Jet Propulsion Laboratory, California Institute of Technology

4800 Oak Grove Dr

Pasadena, CA 91109 USA

[2]Stanford University

[3]California Institute of Technology

[4]Scripps Institution of Oceanography

Abstract—**We have developed the main components of a new flexible and extensible computing environment for geodetic image processing for Synthetic Aperture Radar (SAR) sensors to enable scientists to reduce measurements directly from Level-0 or Level-1 radar data up to Level-3 data products. The Interferometric SAR (InSAR) Scientific Computing Environment (ISCE) can process data from the ALOS, ERS, EnviSAT, Cosmo-SkyMed, RadarSAT-1, RadarSAT-2, and TerraSAR-X platforms; with its flexible design, it can be extended with raw/meta data parsers to enable it to work with radar data from other platforms. The NRC Decadal Survey-recommended DESDynI mission would deliver directly to the science community data of unprecedented quantity and quality, making possible global-scale studies in climate research, natural hazards, and Earth's ecosystem. Applied to a global data set such as from DESDynI, ISCE would enable a new class of analyses at time and spatial scales unavailable using current approaches.**

ISCE is an accurate, extensible, and modular processing system that i) enables multi-scene analysis by adding new algorithms, ii) permits user-reconfigurable operation and extensibility, and iii) capitalizes on codes already developed by NASA and the science community. The framework incorporates modern programming methods, including rigorous componentization of processing codes, abstraction and generalization of data models, and a robust, intuitive user interface with graduated exposure to the levels of sophistication, allowing novices to apply it readily for common tasks and experienced users to mine data with great facility and flexibility. The framework is designed to easily allow user contributions, creating an environment that can extend the framework into the indefinite future.

I. INTRODUCTION

The objectives of the InSAR Scientific Computing Environment (ISCE) are to develop an open, modular, extensible InSAR computing environment for the research community. The environment incorporates state-of-the-art, highly accurate algorithms to facilitate InSAR processing for non-experts and experts alike. To service the community and promote use, the project will deliver documented algorithms, formats and interfaces. The specific goal is to create a code suite that the InSAR community embraces and grows with.

The goals of ISCE are a direct response to the priorities set by an international community of radar processor developers and users as determined by two NASA sponsored InSAR workshops, one convened in 2008 at Stanford University and the other in 2011 at the Scripps Institution of Oceanography. The goals of the 2008 workshop were to assess the strengths and weaknesses of the existing InSAR processing packages at the time, define the capabilities of the next-generation processors required by the user community, and set the standards and structure for new InSAR processor development. The top requirements for the next generation radar processing package coming out of the 2008 workshop were the following: (1) precise and well-characterized products; (2) flexible and extensible modular code to encourage modification and improvement by the user community; (3) and a comprehensive set of user documentation.

The NASA Earth Science Technology Office funded the ISCE project through the Advanced Information Systems Technology (AIST) program to implement these recommendations. The approach our multi-institutional team took was relatively straightforward for a software development project, but perhaps more structured than a typical research code development. We approached the development by applying modern software system engineering techniques and tools for configuration management and maintenance. We first collected community-based requirements for InSAR processing methods and generalized data models, primarily through the 2008 workshop final report. We then used these requirements to define an object-oriented framework. With the framework in place, we populated it with processing modules. Along the way, we are creating documentation of the framework, modules, and use cases.

II. THE ISCE ARCHITECTURE

The ISCE architecture was driven by the following key design principles:

1. Preserve the vast expertise and testing currently encoded in Legacy Software
2. Make that Legacy Software more lean in terms of the number of auxiliary tasks it needs to do (such as self configuration and I/O configuration).
3. Build modern object oriented structures around and behind the legacy code to manage that code and push rather than pull user configuration onto that code before executing that code
4. Implement common functions and services such as I/O through APIs to allow their implementations to change and to allow for user configuration and selection of those functions at run time
5. Build in polymorphism mechanisms to allow user selections to alter the implementations of major processing steps and common functions.

At the core of the ISCE architecture is two legacy InSAR processing packages: ROI_PAC (Rosen *et al.*, 2004) and

STD_PROC (Zebker *et al.,* 2010). Both of these software packages are primarily written in Fortran – mostly using the Fortran 77 version of the language, with some of the transitional features leading up to Fortran 90 such as structures and occasionally dynamic memory allocation – with some of the programs written in C and with scripts written in Perl (ROI_PAC) or Python (STD_PROC). ROI_PAC was initially developed over a decade ago and has been used extensively by the science community to process InSAR data from several different international space borne radar platforms such as ERS, EnviSAT, RadarSAT-1, JERS, ALOS, and TerraSAR-X. STD_PROC is currently being developed at Stanford University and is based on advances in the processing algorithms that came from processors developed for SRTM and UAVSAR. Although STD_PROC is new, we refer to it along with ROI_PAC as legacy code because of its pedigree, and because like ROI_PAC it is received as domain expert software whose functionality we wish to preserve. Also both are written in a similar style that is very effective at accomplishing the processing steps but neither easy for non-experts to use nor very flexible or extensible for expert developers to work with. The ISCE architecture seeks to inject some modern software principles that allow for easier use and greater flexibility and extensibility.

Components

To accomplish these goals, we componentize the legacy code by surrounding them with structures that deliver services to the legacy programs, users and developer. The services replace legacy code interactions with the external world that are handled using mostly primitive language features. The structures that deliver these services are quite different, requiring legacy code modifications to add new "wiring" to receive those services.

Figure 1 shows the architecture of a component that has an embedded legacy core. The processing components are built from framework components and properties through either class inheritance or composition. Configuration and control parameters flow from a controlling or driving application at the top into the component initialization method. The configuration and control parameters are derived from user inputs, either from the command line or from input files, and defaults defined in preferences files or within the application itself. The component itself may also define defaults for parameters. Defaults can always be overridden by user inputs.

The components, applications, and other support software involve a mixture of different programming languages and styles. This multilingual structure requires proper use of application program interfaces (APIs) for effective cooperation among languages. The ISCE architecture preserves legacy radar processing software at the core of each of our components, as shown in Figure 1. The component core is not the raw Fortran program, but rather the essential processing engine with data flow execution management removed. We use C/C++ as the intermediary between Python and Fortran because there is a standard Python API that allows Python and C programs to interact. The C intermediary is referred to as a binding.

A component by itself does not actually do anything. It must be instantiated in another type of component called an *application* that has the responsibility of collecting the user inputs and of managing its components from their initialization to the flow of data through them to their finalization.

Software

The structure of the ISCE software system is shown in Figure 2. At JPL, the software configuration is managed using SVN, and "checking out" the software yields the directory structure shown in the figure. To build the software, the user runs SCons, a Python-based build system, rather than the "configure; make" method. SCons works well building code written in multiple languages, and is essentially Python-programmable to handle a range of build situations can occur on disparate operating system configurations.

The mainline ISCE applications and components are contained under the Applications and Packages directories. Packages are collections of logically related Components, Legacy Cores, and other support software. Current Packages include: *iscesys*, which contains the ISCE system or framework components and properties as well as several APIs; *isceobj,* which contains class definitions for several objects used by the components*; mroipac,* which contains the recasting of ROI_PAC into components; and *stdproc*, which contains the recasting of STD_PROC into components. Figure 2 shows a branch called Contrib, where contributed software can be located without intermingling with the ISCE distribution.

Polymorphism

An ISCE developer or a user will be able to add software objects to the framework after the framework is built and user

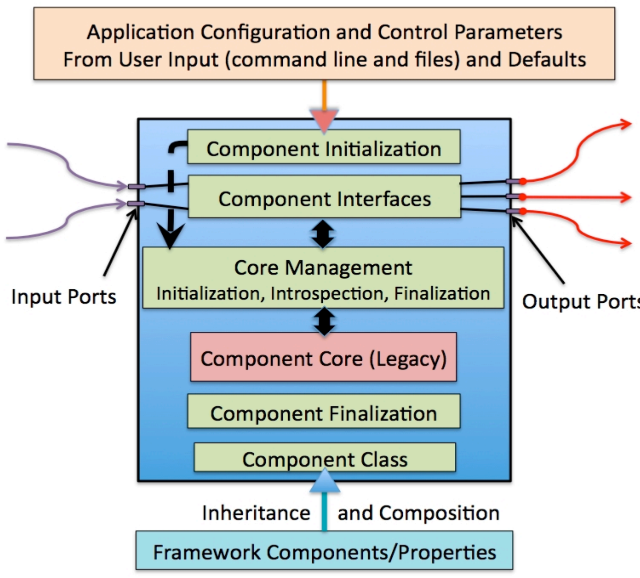
Application Configuration and Control Parameters
From User Input (command line and files) and Defaults
Component Initialization
Component Interfaces
Core Management
Initialization, Introspection, Finalization
Component Core (Legacy)
Component Finalization
Component Class
Input Ports
Output Ports
Inheritance
and Composition
Framework Components/Properties

Figure 1. Architecture of a component

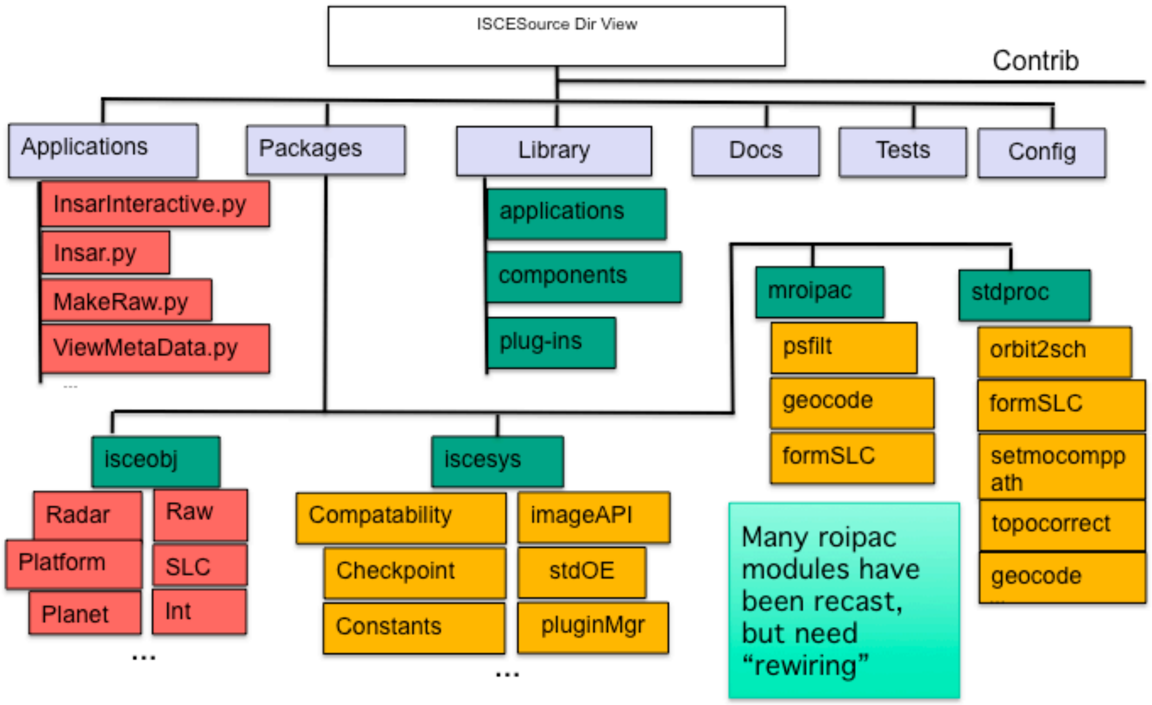

Figure 2. The structure of the ISCE source directory.

inputs or default settings can select the new or the old objects as long as the objects adhere to the correct interfaces. This dynamic alteration of the software for a processing run is possible using object-oriented *polymorphism* design patterns and greatly enhances the flexibility and extensibility of ISCE. We are allowing for two types of polymorphism: (1) *facility* polymorphism where major components may be morphed at run-time; and (2) a *plug-in* type of polymorphism where lower level, common functions such as implementations of fast Fourier transforms (FFTs) may be selected across the board at run-time. Facilities define a task and an interface that are implemented by a component. Registering a Component as a Facility indicates the Component as the default Component to implement the Facility but also alerts the Application to allow the User to specify an alternate Component at runtime to implement the Facility.

Provenance

The ability to log and query the pedigree of a particular piece of processed data, known as "provenance," is important to the community. Provenance allows users to keep track of: the versions of applications, components, and other software that were used to produce a data product; the configuration parameters used to initialize those applications and components; the input data and other output data products at the time of creation of the data product of interest. Provenance enables an investigator to explore data by using different versions of the software or iteratively tweaking parameters, while keeping a record of what was done at every step. This record allows the user or colleagues to reproduce results exactly, by sharing scripts with the community, facilitating reproducible collaborations and publications.

ISCE supports provenance through database management and logging of processing steps and meta-data at each step of the processing chain. Given the python-based object-oriented methods in ISCE, the code lends itself to being used within software packages with higher levels of sophistication that provide provenance capability as well. For example, several GUI interfaces, such as VisTrails, have complete provenance management, and easily accept python applications and plug-ins as modules. This effectively extends the ISCE utility as a scientific tool with essentially no effort.

III. DESCRIPTION OF THE API SOFTWARE

The ISCE framework contains various elements that support the ISCE component architecture. We have coded, documented, and tested the key framework Application Program Interfaces (APIs) that allow us to control processing flow among ISCE modules. These APIs are the following: Image API, Control API, and StdOE API.

Image API

The image API provides a set of library functions that provide the legacy software and new programs developed by users with a reliable and versatile way of performing input and output operations on images. The image API consists of a set of C++ classes that contain an abstraction of a real world image as well as concrete methods to access data from sources (such as, but not limited to, files on disc) and a memory buffer to hold a given portion of an image that can be passed between the C++ and Fortran programs. The C++ classes allow for very general and flexible configuration of the objects instantiated from them without specific regard for the types of images and memory buffer specifications currently in the Fortran programs of ROI_PAC and STD_PROC.

We exploit class inheritance mechanisms, so that new data accessor methods can be layered on top of those currently available without rewriting code that currently works. The code has built-in flexible methods for sequential access to full lines of data, random access by line number, and single pixel access, supporting a number of band interleaving schemes. The Image API handles efficient conversion from one scheme to another on input and output. Machine dependent internal representations of the numerical values are converted on the fly so that data files created on one machine can be used in our software without first creating a new file conforming to the internal representation of binary data on the machine that is running the ISCE software. The generality of the Image API permits fast and efficient cached I/O.

Control API

Generally speaking, modules contain parameters or attributes that need to be set appropriately before they can perform their function. The control API is a set of classes, features and methodologies providing an easy, reproducible, extensible and reconfigurable way to pass data, and to set and examine attributes through *set* and *get* methods.

All ISCE modules inherit from a ComponentInit base class, which allows the initialization of the parameters of the subclass that inherits it by passing an initializer object to it. The ISCE framework provides a set of default initializers that permit initialization from file, from a dictionary (an object consisting of a set of (key,value) pairs) of from another object. Expert users could provide their own initializers, as long as they conform to the architecture specifications.

The ComponentInit class provides a set of methods to allow the user to explore how to use a component, to debug his usage of the component, and to document the state of the control parameters through the following built-in capabilities: (1) determine which variables in the module must be set by the controlling program (i.e., those parameters that have no valid default value); (2) determine which variables have default values and what those default values are so that the user may have the option to override the default values; and (3) render the state of the component to a configurable destination such as to a file or to standard output (i.e., dump the variables and associated documentation of an object to a specified destination that may be used and stored by the user to debug a component or to document the provenance information on the component's state).

Another component of the Control API is the Checkpoint class. The process of check-pointing allows the user to save the state of the system at a given point during the program flow, such that the program can be resumed at a given checkpoint without having to recompute the previous stages. This feature is important in the event that the process was interrupted due to an anomaly in the processing or the data, or because the user chooses to use a different set of processing parameters for processing past a certain point. For example, a user may prefer heavier filtering of the interferogram than the default after seeing the quality of the default result. For this, there is no need to return to any processing done prior to the formation of the interferogram.

StdOE API

An essential element of any program suite is the ability to print informative messages about the status of the processing (e.g. percent completion, derived parameters that may be of interest to the user, etc.) and any error messages that occur. In conventional programming, particularly, in Fortran, coders insert "write" statements into their code that are sometimes compiler dependent, such that when compilers change, all the code needs to be updated. To avoid such tight and brittle connection of logging messages to the code elements, we have created a logging API we call StdOE for "Standard Output and Error." The StdOE consists in a C++ static class used for reconfigurable standard output and error. With this API, it is possible to choose destinations for standard output and standard error messages. For example the user might choose to select the terminal window for both of these output streams or instead send them to separate files, to a network socket, or to the input of some other process.

IV. DESCRIPTION OF THE NEW CORE PROCESSING SOFTWARE

Repeat pass InSAR forms topographic and displacement maps from radar data collected at two different times by a platform such as an orbiting spacecraft. The platform collects data from the same location on Earth at times t1 and t2 as illustrated in Figure 3. The spacecraft position at t2 is separated from its position at t1 by a vector ***B***, which is the interferometric baseline. The typical processing flow is as illustrated in Figure 4. InSAR processing depends on precise knowledge of the position and velocity of the platform at the time of observation of a pixel at a given range and Doppler relative to the platform.

The STD_PROC processor at the core of ISCE preserves the accuracy of its data products by taking advantage of the improved accuracy of orbit determination now available and implementing all of the code in a uniform geometric framework (Zebker *et al*., 2010). This approach, based on well-known motion compensation techniques, also facilitates analysis of a time series of many observations of a particular location on Earth by its use of a motion-compensated geodetic

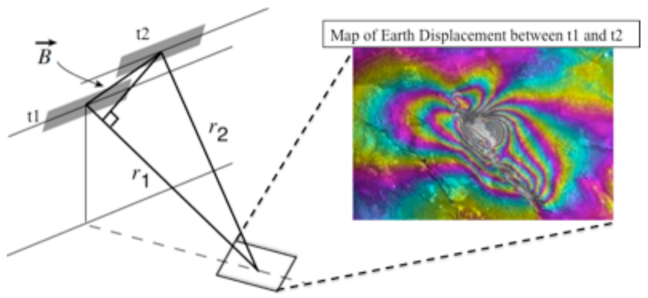

Figure 3. Repeat pass InSAR

coordinate system rather than the traditional range/Doppler coordinate system specific to a given observation, used by for example ROI_PAC. The coordinate system, referred to as SCH (in which S is the local along track direction, C is the cross track direction, and H is the height above the approximating sphere), is based on a local spherical approximation of the ellipsoidal Earth. The equations implemented in the processor are simplified by use of a spherical earth and a corresponding circular approximation of the platform orbit. Figure 5 illustrates the motion compensation geometry, showing the correspondence of an image pixel location, actual satellite position and the assumed reference position on the circular orbit. In this approach, the direction from the satellite to a point on the Earth is determined through the estimated Doppler centroid. Using the motion of the satellite on its orbit relative to the reference track, the position of the satellite is compensated along this direction back to the reference track as a range correction. A corresponding phase correction is also applied. The equations encoded in STD_PROC and hence ISCE are developed in detail in Zebker *et al.* (2010) and Gurrola *et al* (2010).

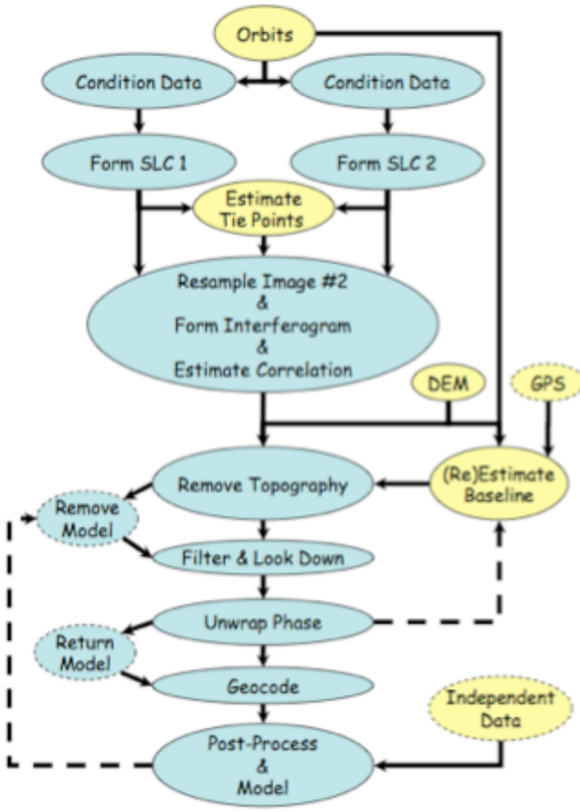

Figure 4. InSAR Processing flow.

V. STATUS OF ISCE DEVELOPMENT

The AIST-sponsored ISCE project has completed two years of development, with one year remaining. The team to date has succeeded in developing all the basic elements of the framework as described in Figs 3 and 5; essentially all the standard ROI_PAC algorithm functionality now exists, but in a modern framework that will serve as a springboard for added capability and community involvement.

To begin engaging the community, a follow-on to the 2008 workshop was held in March 2011 at Scripps Institution of Oceanography. The goals of the 2011 workshop were to compare and validate the accuracy and performance of the various non-commercial InSAR processing packages, both legacy and new, and to generate feedback and suggestions for further developments. Products from four different InSAR packages, including ISCE, were compared for several challenging data sets and recommendations for further comparisons were made, including incorporation of a few commercial InSAR packages and simulated data to isolate differences in results found between the different packages. ISCE did well in terms of accuracy, speed, and ease of use. Some found it difficult to install. The use of Python and the gcc requires a self-consistency to the development environment that many users do not understand. One of the key actions for the ISCE team is to provide more complete descriptions of what constitutes a self-consistent environment, and provide the tools and information for a user to easily create one. One of the top action items from the workshop was to make ISCE available for rapid and wide distribution, indicating that the development is of interest to the community.

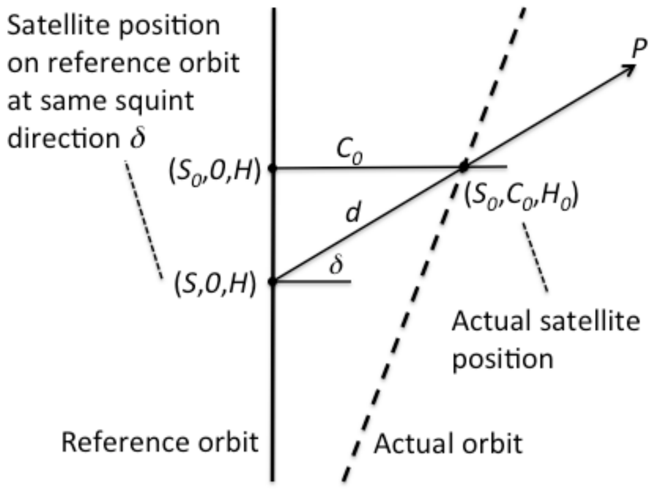
Satellite position on reference orbit at same squint direction δ
P
C₀
(S₀,0,H)
d
(S₀,C₀,H₀)
δ
(S,0,H)
Actual satellite position
Reference orbit
Actual orbit

Figure 5. Motion compensation geometry. The actual satellite position is indicated at coordinate (S_0,C_0,H_0) and its assigned position on the circular reference orbit is at (S,0,H) as determined from the look direction to the imaged pixel at P.

VI. CONCLUSION

There will be an enormous amount of radar data available to the research community in the coming years as more and more space borne international radar systems are launched and the data become available for use, either in real time or from historical archives. With the upcoming launch of the

European Sentinel-1 spacecraft alone, there will be a great deal of data that will be useful for repeat pass interferometric SAR processing. Current tools are nearly all geared to examining individual frames or areas, and are not general enough or generalizable enough to allow researchers to explore the richness of the data in space and time. The ISCE framework and radar processing software associated with it, described in this paper, are designed to be extensible and flexible enough to allow the researcher to ask questions and formulate new ways to answer them with relative ease in the environment.

We have released the software for beta testing to a limited number of users and plan to make wider releases in the coming year as we work through the process of obtaining a suitable license for open release to the research community. We anticipate a documented and usable set of code in the coming year, with ample time for community feedback, bug fixes, and refinement during the remainder of the funded AIST development. And of course, we anticipate that the code will be sufficiently useful, well documented, and accessible for the ISCE to far outlive the duration of the AIST program.

VII. ACKNOWLEDGMENT

We thank Albert Chen and Cody Wortham, both PhD candidates at Stanford University, and Piyush Shanker, PostDoc at Caltech, formerly PhD student at Stanford, for their contributions to the Stanford legacy code at the core of ISCE. The authors would like to thank the Earth Science Technology Office at NASA for support. This work was performed at the Jet Propulsion Laboratory, California Institute of Technology under a contract with NASA, and at Stanford University under a contract with JPL.

VIII. REFERENCES

Gurrola, E., P. Rosen, G. Sacco, W. Szeliga, H. Zebker, M. Simons, D. Sandwell, P. Shanker, C. Wortham, and A. Chen (2010). "InSAR Scientific Computing Environment". 2010 American Geophysical Union Meeting.

Rosen, P. A., S. Hensley, and G. Peltzer (2004), Updated Repeat Orbit Interferometry Package released, Eos Trans. AGU, 85(5).

Rosen, P. *et al.* (2009). "InSAR Scientific Computing Environment". 2009 American Geophysical Union Meeting. Also presented a summary of InSAR SCE and progress at the WinSAR meeting at Fall AGU 2009

Zebker, H., S. Hensley, P. Shanker, C. Wortham (2010). Geodetically Accurate InSAR Data Processor. *IEEE Trans. On Geoscience and Remote Sensing*, 48(12).